IN THE UNITED STATES PATENT AND TRADEMARK OFFICE
BEFORE THE BOARD OF PATENT APPEALS AND INTERFERENCES

	ATTY. DOCKET NO.:	**CH920000018US1**
IN RE APPLICATION OF:	§	
	§	
MICHAEL BAENTSCH, ET AL.	§	**EXAMINER: PATEL, NIRAV B.**
	§	
SERIAL NO.: 09/992,984	§	**CONFIRMATION NO.: 7655**
	§	
FILED: 05 NOVEMBER 2001	§	**ART UNIT: 2135**
	§	
FOR: FILE LANGUAGE VERIFICATION	§	
	§	

APPEAL BRIEF UNDER 37 C.F.R. 41.37

Mail Stop Appeal Briefs - Patents
Commissioner for Patents
P.O. Box 1450
Alexandria, Virginia 22313-1450

Sir:

This Brief is submitted in support of the Appeal of the Examiner's final rejection of Claims 1, 3-6, 10, 12, 14-17 and 19-22 in the above-identified application. A Notice of Appeal was filed in this case on June 25, 2007 and received in the United States Patent and Trademark Office on June 25, 2007. Please charge the fee of $500.00 due under 37 C.F.R. §1.17(c) for filing the brief, as well as any additional required fees, to **IBM CORPORATION DEPOSIT ACCOUNT No. 09-0461**.

REAL PARTY IN INTEREST

The real party in interest in the present Application is International Business Machines Corporation, the Assignee of the present application as evidenced by the Assignment set forth at reel 012578, frame 0233.

RELATED APPEALS AND INTERFERENCES

There are no other appeals or interferences known to Appellants, the Appellants' legal representative, or assignee, which directly affect or would be directly affected by or have a bearing on the Board's decision in the pending appeal.

STATUS OF CLAIMS

Claims 1, 3-6, 10, 12, 14-17 and 19-22 stand finally rejected by the Examiner as noted in the Final Office Action dated April 6, 2007. The rejection of Claims 1, 3-6, 10, 12, 14-17 and 19 under 35 U.S.C. § 103(a) are appealed.

STATUS OF AMENDMENTS

No amendments to the claims have been made subsequent to the April 6, 2007 Final Office Action from which this Appeal is filed.

SUMMARY OF THE CLAIMED SUBJECT MATTER

With regards to **Claim 1**, the present invention is a method for language verification of a Java card CAP file created from an original Java code file (as supported in the originally filed specification on page 28, lines 1-2). The method includes the steps of a) a conversion step for converting said Java card CAP file into a corresponding converted Java code file that is semantically identical to said Java card CAP file (supported on page 4, lines 23-24), wherein the Java card CAP file is created from an the original file contains classes that are capable of being compiled (supported on page 2, lines 16-25), and wherein the only executable instructions in the Java card CAP file are applets (supported on page 2, line 26 to page 3, line 8). The conversion step further includes: a preconversion substep for converting Java card IDs contained in said Java card CAP file into symbolic names, and for converting said Java card CAP file into a standard

Java format, to obtain a preconverted file (supported on page 14, lines 1-5); and a mapping substep for replacing in said preconverted file externally defined names with original names by using a mapping scheme between Java names and tokenized identifiers, to obtain the converted Java code file for a language-verification step (supported on page 14, lines 9-29). The method also includes the step of b) a language-verification step for verifying said converted Java code file for compliance with Java language specifications (supported on page 4, lines 24-26).

With regards to **Claim 3**, the present invention includes the elements of Claim 1, plus the feature of the mapping substep being performed by using a referenced Java export file which is available as a result of creating said Java card CAP file from said original Java code file (as supported by page 14, lines 26-27 of the originally filed specification).

With regards to **Claim 5**, the present invention includes the elements of Claim 4, plus d) a loading step for loading the cryptographic signature file to a chipcard together with the Java card CAP file, wherein the cryptographic signature file is attached to the Java card CAP file when loaded in the chipcard, as supported in the original specification on page 8, lines 7-27 and page 13, lines 16-17.

With regards to **Claim 10**, the present invention describes a computer-readable medium embodying computer program code (supported on page 20, lines 5-20). The computer program code comprises computer executable instructions configured for: converting said Java card CAP file into a corresponding converted Java code file that is semantically identical to said Java card CAP file (supported on page 4, lines 23-24 of the originally filed specification); verifying said converted Java code file for compliance with Java language specifications (supported on page 4, lines 24-26); converting Java card IDs contained in said Java card CAP file into symbolic names (supported on page 4, lines 23-24; converting said Java card CAP file into a standard Java format, to obtain a preconverted file (supported on page 14, lines 1-5); and replacing in said preconverted file externally defined names with original names by using a mapping scheme between Java names and tokenized identifiers, to obtain the converted Java code file (supported on page 14, lines 9-29).

With regards to **Claim 12**, the present invention describes a computer-readable medium (supported on page 20, lines 5-20) containing computer program code for a Java card CAP file language verifier for verifying a Java card CAP file that has been derived from an original Java code file, said Java card CAP file including original Java semantics of said original Java card file (see abstract), the computer program code comprising instructions for: a converter for converting said Java card CAP file into a corresponding converted Java code file that is semantically identical to said Java card CAP file (supported on page 4, lines 23-24), wherein said converter further includes: a preconverter for converting Java card IDs contained in said Java card CAP file into symbolic names, and for converting said Java card CAP file into a standard Java format, to obtain a preconverted file (supported on page 14, lines 1-5); and a mapper for replacing in said preconverted file externally defined names with original names under use of a mapping scheme, to obtain the converted Java code file (supported on page 14, lines 9-29); and a language verifier for verifying said converted Java code file upon its compliance with a Java language specification (supported on page 4, lines 24-26).

With regards to **Claim 14**, the present invention describes the computer-readable medium according to Claim 12, wherein the mapper comprises an input for receiving a referenced Java export file created when a referenced Java card CAP file was converted from its corresponding original Java code file (supported on page 14, lines 9-29).

With regards to **Claim 15**, the present invention describes the computer-readable medium of Claim 12, wherein the instructions are further configured for a signature generator for generating a second cryptographic signature file (supported on page 13, lines 12-13).

With regards to **Claim 16**, the present invention describes the computer-readable medium of Claim 15, wherein the instructions are further configured for loading the second cryptographic signature file and the Java card CAP file to a storage device (supported on page 13, lines 22-23).

With regards to **Claim 17**, the present invention describes a computer-readable medium (supported on page 20, lines 5-20) containing computer program code for a reduced file language verifier for verifying a reduced file that has been converted from an original file, the reduced file

maintaining original semantics of the original file (see abstract), the computer program code comprising instructions for: a converter for converting said reduced file into a corresponding converted file that is semantically identical to said reduced file (supported on page 4, lines 23-24), wherein said converter further includes: a preconverter for converting IDs contained in said reduced file into symbolic names and for converting said reduced file into a standard format, to obtain a preconverted file (supported on page 14, lines 1-5); and a mapper for replacing in said preconverted file externally defined names with original names under use of a mapping scheme, to obtain the converted file (supported on page 14, lines 9-29); means for determining whether said reduced file complies with a predetermined language specification (supported on page 13, lines 1-5); and a language verifier for verifying said converted file upon compliance with the predetermined language specification (supported on page 4, lines 24-26).

With regards to **Claim 19**, the present invention describes the computer-readable medium of Claim 17, wherein said mapper comprises an input for a referenced difference file which is available as a result from a conversion in which a referenced reduced file has been converted from its original file (supported on page 19, lines 5-14).

GROUNDS OF REJECTION TO BE REVIEWED ON APPEAL

A. The Examiner's rejection of Claim 1, 4 and 6 under 35 USC 103(a) as being unpatentable over *Schwabe.* (U.S. Patent No. 6,883,163 – "*Schwabe*") in view of *Stammers et al.* (U.S. Patent No. 7,069,554 – "*Stammers*") is to be reviewed on Appeal.

B. The Examiner's rejection of Claim 3 and 14 under 35 USC 103(a) as being unpatentable over *Schwabe.* (U.S. Patent No. 6,883,163 – "*Schwabe*") in view of *Stammers et al.* (U.S. Patent No. 7,069,554 – "*Stammers*") is to be reviewed on Appeal.

C. The Examiner's rejection of Claim 5 under 35 USC 103(a) as being unpatentable over *Schwabe.* (U.S. Patent No. 6,883,163 – "*Schwabe*") in view of *Stammers et al.* (U.S. Patent No. 7,069,554 – "*Stammers*") and in view of *Ji* (U.S. Patent No. 6,272,641 – "*Ji*")

and in view of *Levy et al.* (U.S. Patent No. 6,092,147 – "*Levy*") is to be reviewed on Appeal.

D. The Examiner's rejection of Claim 10 under 35 USC 103(a) as being unpatentable over *Schwabe.* (U.S. Patent No. 6,883,163 – "*Schwabe*") in view of *Stammers et al.* (U.S. Patent No. 7,069,554 – "*Stammers*") is to be reviewed on Appeal.

E. The Examiner's rejection of Claim 12 and 15 under 35 USC 103(a) as being unpatentable over *Schwabe.* (U.S. Patent No. 6,883,163 – "*Schwabe*") in view of *Stammers et al.* (U.S. Patent No. 7,069,554 – "*Stammers*") is to be reviewed on Appeal.

F. The Examiner's rejection of Claim 16 under 35 USC 103(a) as being unpatentable over *Schwabe.* (U.S. Patent No. 6,883,163 – "*Schwabe*") in view of *Stammers et al.* (U.S. Patent No. 7,069,554 – "*Stammers*") is to be reviewed on Appeal.

G. The Examiner's rejection of Claim 17 and 19 under 35 USC 103(a) as being unpatentable over *Schwabe.* (U.S. Patent No. 6,883,163 – "*Schwabe*") in view of *Stammers et al.* (U.S. Patent No. 7,069,554 – "*Stammers*") is to be reviewed on Appeal.

ARGUMENTS

A. The Examiner's rejection of Claim 1, 4 and 6 under 35 USC 103(a) as being unpatentable over *Schwabe.* (U.S. Patent No. 6,883,163 – "*Schwabe*") in view of *Stammers et al.* (U.S. Patent No. 7,069,554 – "*Stammers*") is to be reviewed on Appeal.

1. The Examiner's rejection of Claim 1, 4 and 6 is improper since the cited prior art does not teach or suggest all of the claimed limitations of the present invention.

A combination of the cited art does not teach or suggest "converting said Java card CAP

file into a corresponding Java code file that is semantically identical to said Java card CAP file, wherein the Java card CAP file is created from an the original file that contains classes that are capable of being compiled" (as supported in the present specification at page 2, lines 16-25), "and wherein the only executable instructions in the Java card CAP file are applets" (as supported on page 2, line 26 to page 3, line 8. That is, the original file includes full function files, while the CAP file only includes applets that must be executed by a browser's Java Virtual Machine (JVM). As described on page 2, line 30 to page 3, line 2 of the present specification, "The virtual machine itself need not load or manipulate Java card CAP files; it need only execute the applet code found in the Java card CAP file that was loaded onto the device by the installation program."

Stammers, and particularly the cited passages, only deals with unpacking Java files from a JAR file. The unpacked (i.e., unzipped) Java files have the same properties as the packed JAR file, in that both include class files. As stated in column 7, lines 37-40 of *Stammers*, "class objects only access other class objects with the same namespace. An object created by a classloader cannot be changed..." There is no teaching or suggestion of two different types of files: 1)standard Java files, and 2) a CAP file that only contains applets.

Furthermore, a combination of the cited art does not teach or suggest "converting Java card Ids contained in said Java card CAP file into symbolic names." The Examiner cites col. 7, lines 21-27 and col. 8, lines 3-7 of *Stammers*, which states:

> The classloader object for the control JAR file thus created can then be used to read classes from the JAR file object created at step S6 and to create objects therefor in the Java virtual machine 32 by converting the class byte code into executable code in Java virtual machine working memory 42. A classloader can create objects only from class files contained in its own signed JAR file.
>
> Referring to FIG. 8, at step S60, the control JAR file classloader created at step S12 (FIG. 5) calls the initialization file object created at step S2 to read the initialization file 50 to locate the platform security level file 56.

There is no teaching or suggestion of card IDs at all, much less IDs that have been converted into symbolic names.

Furthermore, a combination of the cited art does not teach or suggest "replacing said preconverted file externally defined names with original names by using a mapping scheme between Java names and tokenized identifiers, to obtain the converted Java code file for a language-verification step." The Examiner cites col. 7, lines 25-40 of *Stammers* for teaching this feature. The cited passage of *Stammers* states:

> A classloader can create objects only from class files contained in its own signed JAR file. As is well known to the skilled person, every object created by a classloader is tagged with a reference to that classloader. A classloader maintains references to all of the objects it has created in a hash table keyed on the object name. When a class object in the Java virtual machine working memory 42 calls another class object, the Java virtual machine 32 directs the call to the classloader of the calling object. This means that a classloader defines a "namespace" within the Java virtual machine, and is able to regulate the objects loaded into that namespace. A classloader ensures that class objects may only access other class objects within the same namespace. An object created by a classloader cannot be changed by a user of computer 2.

While *Stammers* generally references the use of names, there is no suggestion of replacing "preconverted file externally defines names" using a mapping scheme between Java names and "tokenized identifiers." Rather, token identifiers are never mentioned or suggested at all by *Stammers*.

Furthermore, a combination of the cited art does not teach or suggest "verifying said converted Java code file for compliance with Java language specifications." The Examiner cites col. 16, line 60 to col. 17, line 33 of *Schwabe*, which states:

> Turning now to FIG. 10D, a code sample that illustrates verification that follows an execution path to an API in accordance with one embodiment of the present invention is presented. FIG. 10D includes code samples for a library package L0 800 and applet A1 805. Applet A1 805 includes references to items in package L0 800. FIG. 10D also illustrates a virtual stack before (810) and after (815) execution of source code statements in L0 800 and A1 805. Method A10 820 references method A11 825 at reference numeral 830. Method A11 825 references method L01 835 at reference numeral 840.
>
> Verification of applet A1 805 begins with method A10 820. At 830, method A10 820 invokes method A11 825 with the short integer parameter S 845 and assigns the result to byte array ba 850. In preparation for the method A11 invocation (830), the method A11 825 parameter types are put on the stack 850. As mentioned above, in Java.TM. technology, values of type byte, short and integer

are represented as integer types on the stack. Thus, before invoking method A11 825, the virtual stack 850 contains type int, the type for S 845. This matches the declaration of method A11 825 found in the A1 binary file 805.

At 840, method A11 825 invokes method L01 835 and assigns the result to byte array type ba 855. Before invoking method L01 835, the virtual stack 860 contains a reference to class A1. The expected type is type Object 865. A1 860 is assignment-compatible with Object 865 because A1 860 extends Object (870). This matches the declaration of method L01 835 found in the L0 API definition file 800.

Next, the virtual stack is updated by removing the arguments to the invoked method and adding the return type, if any, of the invoked method. The return type is defined in the referenced API definition file. In the above example, method L01 875 returns an integer type. At 855, the returned integer type is explicitly cast to type byte, which matches the type of ba[0] 880. Method A11 825 returns a byte array, which corresponds to the type of byte array ba 850.

Thus, method A10 820 has been verified without reference to the binary files containing compilation units referenced by method A10 820. Instead, method A10 820 has been verified by examining the content of method A10 820 and the API definition files of all compilation units referenced by method A10 820.

Schwabe, and particularly the cited passage, is directed only to confirming that a method calls a correct type of file. There is no teaching or suggestion of verifying that the code itself (and particularly the converted Java code file) is in compliance with the Java language specifications.

2. The Examiner's rejection of Claim 1, 4 and 6 is improper since there is no motivation to combine features that may be taught in the cited prior art.

Even if the cited art were to be construed as teaching or suggesting all of the limitations found in the Claim 1, there is still no motivation to combine these features.

The proper rationales for arriving at a conclusion of obviousness, as suggested by the U.S. Supreme court in the case of KSR International Co. v. Teleflex, Inc, et al., 127 S. Ct. 1727 (2007), include the following tests for determining a motivation to combined elements from the prior art:

A. Combining prior art elements according to known methods to yield predictable results;

B. Simple substitution of one known element for another to obtain predictable results;

C. Use of a known technique to improve similar devices in a the same way;

D. Applying a known technique to a known device ready for improvement to yield predictable results;

E. "Obvious to try" – choosing from a finite number of identified, predictable solutions, with a reasonable expectation of success;

F. Some teaching, suggestion, or motivation in the prior art that would have led one of ordinary skill to modify the prior art reference or to combine prior art reference teachings to arrive at the claimed invention. (All emphasis added.)

The Examiner has used none of these tests. Rather, the Examiner only offers a conclusory statement that "it would have been obvious to a person of ordinary skill in the art at the time the invention was made to combined Stammers with Schwabe to arrange and test the functionality components, since one would have been motivated to verify the authenticity and/or interaction with the other components [*Stammers*, col. 2, lines 14-16]."

The complete cited passage of *Stammers* states:

> The present invention also provides an apparatus or method in which functional components are arranged and tested to verify their authenticity and/or verify the interaction which is allowed with other components.

This passage, as well as the Examiner's statement, is not germane to the question of motivation. Specifically, *Schwabe* is cited for teaching the creation of a Java card CAP file from an original file, that the Java card CAP file only contains applets, and the language-verification step. *Stammers* is cited for all other features in Claim 1. Thus, the Examiner appears to be stating that *Stammers* provides the motivation to verify authenticity or allowable interaction of functional components, which *Schwabe* fulfils. However, the teaching of *Schwabe* is not directed to verifying authenticity and/or interaction capabilities of components. Thus, neither *Schwabe* nor *Stammers* provides the motivation for utilizing such a component.

For reasons cited above, this rejection is not well founded, and the rejection of Claim 1, 4 and 6 should be reversed.

B. The Examiner's rejection of Claim 3 and 14 under 35 USC 103(a) as being unpatentable over *Schwabe.* (U.S. Patent No. 6,883,163 – "*Schwabe*") in view of *Stammers et al.* (U.S. Patent No. 7,069,554 – "*Stammers*") is to be reviewed on Appeal.

1. The Examiner's rejection of Claim 3 and 14 is improper since the cited prior art does not teach or suggest all of the claimed limitations of the present invention.

The examiner cites col. 18, lines 5-17 and col. 16, lines 18-35 of *Schwabe* for teaching the feature of "wherein said mapping substep is performed using a referenced Java export file which is available as a result of creating said Java card CAP file from said original Java code file." The cited passages state, respectively:

> According to one embodiment of the present invention, the resource-constrained device is a Java Card.TM. enabled device. In this embodiment, the API definition file is Java Card.TM. export file, the binary file is a class file and the optimized binary file is a CAP file. Also, the methods in a class to be loaded are bytecode programs, which when interpreted will result in a series of executable instructions. According to this embodiment, the bytecode program verifier 1020 verifies the integrity of the bytecode programs in a CAP file with reference to the CAP file, the export file corresponding to the CAP file, and the export file containing externally referenced items. If all the methods are successfully verified, the CAP file is sent to the resource-constrained device 1040 via a terminal device 1045.
>
> Verification using an API definition file according to embodiments of the present invention follows the same four steps shown above with reference to FIG. 10A, except that information about the invoked method is obtained from an API definition file instead of a binary file. The conclusions drawn regarding verification of the referencing binary file are the same in both cases. In addition, at some point during verification, the API definition file is verified for internal consistency. This step is parallel to verifying a referenced binary file. Furthermore, during verification using an API definition file according to embodiments of the present invention, the assumption is made that an implementation of the API definition file has been verified in a previous operation

and that the implementation is complainant with the API definition file. This is described in more detail with reference to FIGS. 10C and 10D.

There is no teaching or suggestion that the Java export file "is available as a result of creating said Java card CAP file." Rather, the Java Card export file is simply an API definition file that is assumed to exist, along with the class file and CAP file.

2. The Examiner's rejection of Claim 3 and 14 is improper since there is no motivation to combine features that may be taught in the cited prior art.

The proper rationales for determining motivation to combine are cited above, and are not reiterated here.

The Examiner never states any rationale for a motivation to combine the teachings of *Schwabe* and *Stammers*. Appellants contend that this silence is due to the lack of such motivation.

For reasons cited above, this rejection is not well founded, and the rejection of Claim 3 and 14 should be reversed.

C. The Examiner's rejection of Claim 5 under 35 USC 103(a) as being unpatentable over *Schwabe*. (U.S. Patent No. 6,883,163 – "*Schwabe*") in view of *Stammers et al.* (U.S. Patent No. 7,069,554 – "*Stammers*") and in view of *Ji* (U.S. Patent No. 6,272,641 – "*Ji*") and in view of *Levy et al.* (U.S. Patent No. 6,092,147 – "*Levy*") is to be reviewed on Appeal.

The Examiner's rejection of Claim 5 is improper since the cited prior art does not teach or suggest all of the claimed limitations of the present invention.

The Examiner states that *Ji* teaches a Java card CAP file with a signature in Figure 2 of *Ji*. Actually, *Ji*, and particularly Figure 2, teaches a scanner. There is no teaching or suggestion

in *Ji* of a Java card CAP file, particularly a Java card CAP file that is loaded along with a cryptographic signature file onto a chipcard.

Furthermore, the Examiner states that *Levy* teaches on col. 6, lines 11-27, the attachment of a cryptographic signature file to "the Java carp CAP file when loaded in the chipcard." The cited passage from *Levy* states:

> The front end bytecode verifier 68 may verify that one or more bytecodes entering the converter from source outside of the converter, such as compilers or other forms of software application generators, conform to a predetermined set of criteria. The criteria may be similar to the verification steps described above with reference to FIG. 2. Any bytecodes which do not conform to the criteria may be rejected. The resulting verified bytecodes may be transferred to the bytecode authenticator 70. The bytecode authenticator may receive bytecodes exclusively from the bytecode front end verifier and may compute and generate a proof of authenticity, as is well known, on the one or more verified bytecodes using on any suitable cryptographic computation. A suitable cryptographic computation may include, for example, a hash value, a message authentication code using a block-cipher algorithm, or a digital signature using an asymmetric cryptographic algorithm.

There is no suggestion, by any combination of the cited art, of attaching a cryptographic signature file to a Java card CAP file when the Java card CAP file is loaded into a chipcard.

Thus, this rejection is not well founded, and the rejection of Claim 5 should be reversed.

D. The Examiner's rejection of Claim 10 under 35 USC 103(a) as being unpatentable over *Schwabe*. (U.S. Patent No. 6,883,163 – "*Schwabe*") in view of *Stammers et al.* (U.S. Patent No. 7,069,554 – "*Stammers*") is to be reviewed on Appeal.

1. The Examiner's rejection of Claim 10 is improper since the cited prior art does not teach or suggest all of the claimed limitations of the present invention.

A combination of the cited art does not teach or suggest "converting said Java card CAP file into a corresponding Java code file that is semantically identical to said Java card CAP file."

That is, the original file includes full function files, while the CAP file only includes applets that must be executed by a browser's Java Virtual Machine (JVM). As described on page 2, line 30 to page 3, line 2 of the present specification, "The virtual machine itself need not load or manipulate Java card CAP files; it need only execute the applet code found in the Java card CAP file that was loaded onto the device by the installation program."

Stammers, and particularly the cited passages, only deals with unpacking Java files from a JAR file. The unpacked (i.e., unzipped) Java files have the same properties as the packed JAR file, in that both include class files. As stated in column 7, lines 37-40 of *Stammers*, "class objects only access other class objects with the same namespace. An object created by a classloader cannot be changed..."

Furthermore, a combination of the cited art does not teach or suggest "converting Java card IDs contained in said Java card CAP file into symbolic names." The Examiner cites col. 7, lines 21-27 and col. 8, lines 3-7 of *Stammers*, which states:

> The classloader object for the control JAR file thus created can then be used to read classes from the JAR file object created at step S6 and to create objects therefor in the Java virtual machine 32 by converting the class byte code into executable code in Java virtual machine working memory 42. A classloader can create objects only from class files contained in its own signed JAR file.
>
> Referring to FIG. 8, at step S60, the control JAR file classloader created at step S12 (FIG. 5) calls the initialization file object created at step S2 to read the initialization file 50 to locate the platform security level file 56.

There is no teaching or suggestion of card IDs at all, much less IDs that have been converted into symbolic names.

Furthermore, a combination of the cited art does not teach or suggest "replacing said preconverted file externally defined names with original names by using a mapping scheme between Java names and tokenized identifiers, to obtain the converted Java code file for a language-verification step." The Examiner cites col. 7, lines 25-40 of *Stammers* for teaching this feature. The cited passage of *Stammers* states:

A classloader can create objects only from class files contained in its own signed

> JAR file. As is well known to the skilled person, every object created by a classloader is tagged with a reference to that classloader. A classloader maintains references to all of the objects it has created in a hashtable keyed on the object name. When a class object in the Java virtual machine working memory 42 calls another class object, the Java virtual machine 32 directs the call to the classloader of the calling object. This means that a classloader defines a "namespace" within the Java virtual machine, and is able to regulate the objects loaded into that namespace. A classloader ensures that class objects may only access other class objects within the same namespace. An object created by a classloader cannot be changed by a user of computer 2.

While *Stammers* generally references the use of names, there is no suggestion of replacing "preconverted file externally defines names" using a mapping scheme between Java names and "tokenized identifiers." Rather, token identifiers are never mentioned or suggested at all by *Stammers*.

Furthermore, a combination of the cited art does not teach or suggest "verifying said converted Java code file for compliance with Java language specifications." The Examiner cites col. 16, line 60 to col. 17, line 33 of *Schwabe*, which states:

> Turning now to FIG. 10D, a code sample that illustrates verification that follows an execution path to an API in accordance with one embodiment of the present invention is presented. FIG. 10D includes code samples for a library package L0 800 and applet A1 805. Applet A1 805 includes references to items in package L0 800. FIG. 10D also illustrates a virtual stack before (810) and after (815) execution of source code statements in L0 800 and A1 805. Method A10 820 references method A11 825 at reference numeral 830. Method A11 825 references method L01 835 at reference numeral 840.
>
> Verification of applet A1 805 begins with method A10 820. At 830, method A10 820 invokes method A11 825 with the short integer parameter S 845 and assigns the result to byte array ba 850. In preparation for the method A11 invocation (830), the method A11 825 parameter types are put on the stack 850. As mentioned above, in Java.TM. technology, values of type byte, short and integer are represented as integer types on the stack. Thus, before invoking method A11 825, the virtual stack 850 contains type int, the type for S 845. This matches the declaration of method A11 825 found in the A1 binary file 805.
>
> At 840, method A11 825 invokes method L01 835 and assigns the result to byte array type ba 855. Before invoking method L01 835, the virtual stack 860 contains a reference to class A1. The expected type is type Object 865. A1 860 is assignment-compatible with Object 865 because A1 860 extends Object (870).

This matches the declaration of method L01 835 found in the L0 API definition file 800.

Next, the virtual stack is updated by removing the arguments to the invoked method and adding the return type, if any, of the invoked method. The return type is defined in the referenced API definition file. In the above example, method L01 875 returns an integer type. At 855, the returned integer type is explicitly cast to type byte, which matches the type of ba[0] 880. Method A11 825 returns a byte array, which corresponds to the type of byte array ba 850.

Thus, method A10 820 has been verified without reference to the binary files containing compilation units referenced by method A10 820. Instead, method A10 820 has been verified by examining the content of method A10 820 and the API definition files of all compilation units referenced by method A10 820.

The Examiner also cites *Schwabe* at col. 16, lines 18-35, and col. 18, lines 5-17 which state:

Verification using an API definition file according to embodiments of the present invention follows the same four steps shown above with reference to FIG. 10A, except that information about the invoked method is obtained from an API definition file instead of a binary file. The conclusions drawn regarding verification of the referencing binary file are the same in both cases. In addition, at some point during verification, the API definition file is verified for internal consistency. This step is parallel to verifying a referenced binary file. Furthermore, during verification using an API definition file according to embodiments of the present invention, the assumption is made that an implementation of the API definition file has been verified in a previous operation and that the implementation is complainant with the API definition file. This is described in more detail with reference to FIGS. 10C and 10D.

According to one embodiment of the present invention, the resource-constrained device is a Java Card.TM. enabled device. In this embodiment, the API definition file is Java Card.TM. export file, the binary file is a class file and the optimized binary file is a CAP file. Also, the methods in a class to be loaded are bytecode programs, which when interpreted will result in a series of executable instructions. According to this embodiment, the bytecode program verifier 1020 verifies the integrity of the bytecode programs in a CAP file with reference to the CAP file, the export file corresponding to the CAP file, and the export file containing externally referenced items. If all the methods are successfully verified, the CAP file is sent to the resource-constrained device 1040 via a terminal device 1045.

Schwabe, and particularly the cited passages, is directed only to confirming that a method calls a correct type of file. There is no teaching or suggestion of verifying that the code itself (and particularly the converted Java code file) is in compliance with the Java language specifications.

2. The Examiner's rejection of Claim 10 is improper since there is no motivation to combine features that may be taught in the cited prior art.

Even if the cited art were to be construed as teaching or suggesting all of the limitations found in the Claim 1, there is still no motivation to combine these features.

The proper rationales for arriving at a conclusion of obviousness, as suggested by *KSR*, are presented above. Although not expressly stated, it appears from the Examiner's statement "The code sample and the object are similar" indicates that the Examiner is utilizing Option B (simple substitution of one known element for another to obtain predictable results). The code sample being referenced is pseudocode, found in the Figures, for verifying that methods are properly called. Appellants are unclear as to what "object" is similar to the code sample. If it is the Examiner's position that code, which is used to confirm that an object properly calls another object, is the same as verifying that a converted Java code file is in compliance with the Java specification, then this is not a "simple substitution" nor would it lead to "predictable results." That is, replacing code that confirms calls to correct methods with an examination of the schema of the code being run is not a "simple substitution." Only by using the present invention as a template for examining the format of a converted Java code file can one arrive at the present invention, which is prohibited by In re Fritch, 972 F.2d 1260 (Fed.Cir. 1992).

For reasons cited above, this rejection is not well founded, and the rejection of Claim 10 should be reversed.

E. The Examiner's rejection of Claim 12 and 15 under 35 USC 103(a) as being unpatentable over *Schwabe*. (U.S. Patent No. 6,883,163 – "*Schwabe*") in view of *Stammers et al.* (U.S. Patent No. 7,069,554 – "*Stammers*") is to be reviewed on Appeal.

1. The Examiner's rejection of Claim 12 and 15 is improper since the cited prior art does not teach or suggest all of the claimed limitations of the present invention.

Claim 15 depends directly on Claim 12.

A combination of the cited art does not teach or suggest "converting said Java card CAP file into a corresponding Java code file that is semantically identical to said Java card CAP file." That is, the original file includes full function files, while the CAP file only includes applets that must be executed by a browser's Java Virtual Machine (JVM). As described on page 2, line 30 to page 3, line 2 of the present specification, "The virtual machine itself need not load or manipulate Java card CAP files; it need only execute the applet code found in the Java card CAP file that was loaded onto the device by the installation program."

Stammers, and particularly the cited passages, only deals with unpacking Java files from a JAR file. The unpacked (i.e., unzipped) Java files have the same properties as the packed JAR file, in that both include class files. As stated in column 7, lines 37-40 of *Stammers*, "class objects only access other class objects with the same namespace. An object created by a classloader cannot be changed..."

Furthermore, a combination of the cited art does not teach or suggest "converting Java card Ids contained in said Java card CAP file into symbolic names." The Examiner cites col. 7, lines 21-27 and col. 8, lines 3-7 of *Stammers*, which states:

> The classloader object for the control JAR file thus created can then be used to read classes from the JAR file object created at step S6 and to create objects therefor in the Java virtual machine 32 by converting the class byte code into executable code in Java virtual machine working memory 42. A classloader can create objects only from class files contained in its own signed JAR file.
>
> Referring to FIG. 8, at step S60, the control JAR file classloader created at step S12 (FIG. 5) calls the initialization file object created at step S2 to read the initialization file 50 to locate the platform security level file 56.

There is no teaching or suggestion of card IDs at all, much less Ids that have been converted into symbolic names.

Furthermore, a combination of the cited art does not teach or suggest "verifying said converted Java code file for compliance with Java language specifications." The Examiner cites col. 16, line 60 to col. 17, line 33 of *Schwabe*, which states:

> Turning now to FIG. 10D, a code sample that illustrates verification that follows an execution path to an API in accordance with one embodiment of the present invention is presented. FIG. 10D includes code samples for a library package L0 800 and applet A1 805. Applet A1 805 includes references to items in package L0 800. FIG. 10D also illustrates a virtual stack before (810) and after (815) execution of source code statements in L0 800 and A1 805. Method A10 820 references method A11 825 at reference numeral 830. Method A11 825 references method L01 835 at reference numeral 840.
>
> Verification of applet A1 805 begins with method A10 820. At 830, method A10 820 invokes method A11 825 with the short integer parameter S 845 and assigns the result to byte array ba 850. In preparation for the method A11 invocation (830), the method A11 825 parameter types are put on the stack 850. As mentioned above, in Java.TM. technology, values of type byte, short and integer are represented as integer types on the stack. Thus, before invoking method A11 825, the virtual stack 850 contains type int, the type for S 845. This matches the declaration of method A11 825 found in the A1 binary file 805.
>
> At 840, method A11 825 invokes method L01 835 and assigns the result to byte array type ba 855. Before invoking method L01 835, the virtual stack 860 contains a reference to class A1. The expected type is type Object 865. A1 860 is assignment-compatible with Object 865 because A1 860 extends Object (870). This matches the declaration of method L01 835 found in the L0 API definition file 800.
>
> Next, the virtual stack is updated by removing the arguments to the invoked method and adding the return type, if any, of the invoked method. The return type is defined in the referenced API definition file. In the above example, method L01 875 returns an integer type. At 855, the returned integer type is explicitly cast to type byte, which matches the type of ba[0] 880. Method A11 825 returns a byte array, which corresponds to the type of byte array ba 850.
>
> Thus, method A10 820 has been verified without reference to the binary files containing compilation units referenced by method A10 820. Instead, method A10 820 has been verified by examining the content of method A10 820 and the API definition files of all compilation units referenced by method A10 820.

Schwabe, and particularly the cited passage, is directed only to confirming that a method calls a correct type of file. There is no teaching or suggestion of verifying that the code itself (and particularly the converted Java code file) is in compliance with the Java language specifications.

2. The Examiner's rejection of Claim 12 and 15 is improper since there is no motivation to combine features that may be taught in the cited prior art.

Even if the cited art were to be construed as teaching or suggesting all of the limitations found in the Claim 1, there is still no motivation to combine these features.

The proper rationales for arriving at a conclusion of obviousness, as suggested by *KSR*, are presented above. The rejection of Claim 12 has been "lumped together" with the rejection of Claim 10. Thus, it appears from the Examiner's statement that "The code sample and the object are similar" indicates that the Examiner is utilizing Option B (simple substitution of one known element for another to obtain predictable results). The code sample being referenced is pseudocode, found in the Figures, for verifying that methods are properly called. Appellants are unclear as to what "object" is similar to the code sample. If it is the Examiner's position that code, which is used to confirm that an object properly calls another object, is the same as verifying that a converted Java code file is in compliance with the Java specification, then this is not a "simple substitution" nor would it lead to "predictable results." That is, replacing code that confirms calls to correct methods with an examination of the schema of the code being run is not a "simple substitution." Only by using the present invention as a template for examining the format of a converted Java code file can one arrive at the present invention, which is prohibited by In re Fritch, 972 F.2d 1260 (Fed.Cir. 1992).

For reasons cited above, this rejection is not well founded, and the rejection of Claim 12 and 15 should be reversed.

F. The Examiner's rejection of Claim 16 under 35 USC 103(a) as being unpatentable over *Schwabe*. (U.S. Patent No. 6,883,163 – "*Schwabe*") in view of *Stammers et al.* (U.S. Patent No. 7,069,554 – "*Stammers*") is to be reviewed on Appeal.

1. The Examiner's rejection of Claim 16 is improper since the cited prior art does not teach or suggest all of the claimed limitations of the present invention.

The Examiner cites *Schwabe* at col., 17, lines 55-67 and col. 18, lines 15-25 for teaching a "second cryptographic file" that is loaded into a storage device. The cited passages state:

> Client memory 1000 stores: an operating system 1010; a program converter 1015, which converts binary file and related API definition files into optimized binary files and API definition files; a program verifier 1020 for verifying whether or not a specified program satisfies certain predefined integrity criteria; at least one optimized binary file repository 1025, for locally storing optimized binary files in use and/or available for use by users of the computer 1000; at least one API definition file repository 1030 for storing export files.
>
> The converter 1015 converts a binary file into an optimized binary file and an API definition file of the optimized binary file. If the binary file includes external reference, the converter 1015 uses the API definition file stored in 1030 of the module including the external reference to verify the external reference.
>
> According to one embodiment of the present invention, the resource-constrained device is a Java Card.TM. enabled device. In this embodiment, the API definition file is Java Card.TM. export file, the binary file is a class file and the optimized binary file is a CAP file. Also, the methods in a class to be loaded are bytecode programs, which when interpreted will result in a series of executable instructions. According to this embodiment, the bytecode program verifier 1020 verifies the integrity of the bytecode programs in a CAP file with reference to the CAP file, the export file corresponding to the CAP file, and the export file containing externally referenced items. If all the methods are successfully verified, the CAP file is sent to the resource-constrained device 1040 via a terminal device 1045.
>
> As shown in FIG. 11A, a terminal 1045 is equipped with a card acceptance device (CAD) 1050 for receiving a card. The terminal 1045 may be connected to a network 1055 that communicates with a plurality of other computing devices, such as a server 985. It is possible to load data and software onto a smart card over the network 1055 using card equipped devices. Downloads of this nature include applets or libraries to be loaded onto a smart card as well as digital cash and other information used in accordance with a variety of electronic commerce and other applications.

There is no reference to any second signature file at all, much less one that is loaded into a storage device along with the Java card CAP file.

2. The Examiner's rejection of Claim 16 is improper since there is no motivation to combine features that may be taught in the cited prior art.

Even if the cited art were construed to teach or suggest all of the elements in Claim 16, he Examiner fails again to provide any rationale for combining the cited art.

For reasons so stated, this rejection is not well founded, and Claim 16 should be allowed.

G. The Examiner's rejection of Claim 17 and 19 under 35 USC 103(a) as being unpatentable over *Schwabe.* (U.S. Patent No. 6,883,163 – "*Schwabe*") in view of *Stammers et al.* (U.S. Patent No. 7,069,554 – "*Stammers*") is to be reviewed on Appeal.

1. The Examiner's rejection of Claim 17 and 19 is improper since the cited prior art does not teach or suggest all of the claimed limitations of the present invention.

A combination of the cited art does not teach or suggest "converting said reduced files into a corresponding converted file that is semantically identical to said reduced file."

Stammers, and particularly the cited passages, only deals with unpacking Java files from a JAR file. The unpacked (i.e., unzipped) Java files have the same properties as the packed JAR file, in that both include class files. As stated in column 7, lines 37-40 of *Stammers*, "class objects only access other class objects with the same namespace. An object created by a classloader cannot be changed..."

Furthermore, a combination of the cited art does not teach or suggest "converting IDs contained in said reduced file into symbolic names." The Examiner cites col. 7, lines 21-27 and

col. 8, lines 3-7 of *Stammers*, which states:

> The classloader object for the control JAR file thus created can then be used to read classes from the JAR file object created at step S6 and to create objects therefor in the Java virtual machine 32 by converting the class byte code into executable code in Java virtual machine working memory 42. A classloader can create objects only from class files contained in its own signed JAR file.
>
> Referring to FIG. 8, at step S60, the control JAR file classloader created at step S12 (FIG. 5) calls the initialization file object created at step S2 to read the initialization file 50 to locate the platform security level file 56.

There is no teaching or suggestion of IDs at all, much less IDs that have been converted into symbolic names.

Furthermore, a combination of the cited art does not teach or suggest "verifying said converted file for compliance with a predetermined language specifications." The Examiner cites col. 16, line 60 to col. 17, line 33 of *Schwabe*, which states:

> Turning now to FIG. 10D, a code sample that illustrates verification that follows an execution path to an API in accordance with one embodiment of the present invention is presented. FIG. 10D includes code samples for a library package L0 800 and applet A1 805. Applet A1 805 includes references to items in package L0 800. FIG. 10D also illustrates a virtual stack before (810) and after (815) execution of source code statements in L0 800 and A1 805. Method A10 820 references method A11 825 at reference numeral 830. Method A11 825 references method L01 835 at reference numeral 840.
>
> Verification of applet A1 805 begins with method A10 820. At 830, method A10 820 invokes method A11 825 with the short integer parameter S 845 and assigns the result to byte array ba 850. In preparation for the method A11 invocation (830), the method A11 825 parameter types are put on the stack 850. As mentioned above, in Java.TM. technology, values of type byte, short and integer are represented as integer types on the stack. Thus, before invoking method A11 825, the virtual stack 850 contains type int, the type for S 845. This matches the declaration of method A11 825 found in the A1 binary file 805.
>
> At 840, method A11 825 invokes method L01 835 and assigns the result to byte array type ba 855. Before invoking method L01 835, the virtual stack 860 contains a reference to class A1. The expected type is type Object 865. A1 860 is assignment-compatible with Object 865 because A1 860 extends Object (870). This matches the declaration of method L01 835 found in the L0 API definition file 800.

> Next, the virtual stack is updated by removing the arguments to the invoked method and adding the return type, if any, of the invoked method. The return type is defined in the referenced API definition file. In the above example, method L01 875 returns an integer type. At 855, the returned integer type is explicitly cast to type byte, which matches the type of ba[0] 880. Method A11 825 returns a byte array, which corresponds to the type of byte array ba 850.
>
> Thus, method A10 820 has been verified without reference to the binary files containing compilation units referenced by method A10 820. Instead, method A10 820 has been verified by examining the content of method A10 820 and the API definition files of all compilation units referenced by method A10 820.

Schwabe, and particularly the cited passage, is directed only to confirming that a method calls a correct type of file. There is no teaching or suggestion of verifying that the code itself (and particularly the converted code file) is in compliance with a predetermined language specification.

2. The Examiner's rejection of Claim 17 and 19 is improper since there is no motivation to combine features that may be taught in the cited prior art.

Even if the cited art were to be construed as teaching or suggesting all of the limitations found in the Claim 17, there is still no motivation to combine these features.

The proper rationales for arriving at a conclusion of obviousness, as suggested by *KSR*, are presented above. The rejection of Claim 17 has been "lumped together" with the rejection of Claim 10. Thus, it appears from the Examiner's statement that "The code sample and the object are similar" indicates that the Examiner is utilizing Option B (simple substitution of one known element for another to obtain predictable results). The code sample being referenced is pseudocode, found in the Figures, for verifying that methods are properly called. Appellants are unclear as to what "object" is similar to the code sample. If it is the Examiner's position that code, which is used to confirm that an object properly calls another object, is the same as verifying that a converted file is in compliance with a predetermined language's specification, then this is not a "simple substitution" nor would it lead to "predictable results." That is, replacing code that confirms calls to correct methods with an examination of the schema of the code being run is not a "simple substitution." Only by using the present invention as a template

for examining the format of a converted file can one arrive at the present invention, which is prohibited by In re Fritch, 972 F.2d 1260 (Fed.Cir. 1992).

For reasons cited above, this rejection is not well founded, and the rejection of Claim 17 and 19 should be reversed.

CONCLUSION

Appellants have pointed out with specificity the manifest error in the Examiner's rejections, and the claim language which renders the invention patentable over the various combinations of references. Appellants, therefore, respectfully request that this case be remanded to the Examiner with instructions to issue a Notice of Allowance for all pending claims.

Respectfully submitted,

James E. Boice
Reg. No. 44,545
DILLON & YUDELL LLP
8911 N. Capital of Texas Highway
Suite 2110
Austin, Texas 78759
512-343-6116

ATTORNEY FOR APPELLANTS

CLAIMS APPENDIX

1. A method for language verification of a Java card CAP file created from an original Java code file, comprising:

a) a conversion step for converting said Java card CAP file into a corresponding converted Java code file that is semantically identical to said Java card CAP file, wherein the Java card CAP file is created from an the original file contains classes that are capable of being compiled, and wherein the only executable instructions in the Java card CAP file are applets, wherein said conversion step further includes:

a preconversion substep for converting Java card IDs contained in said Java card CAP file into symbolic names, and for converting said Java card CAP file into a standard Java format, to obtain a preconverted file; and

a mapping substep for replacing in said preconverted file externally defined names with original names by using a mapping scheme between Java names and tokenized identifiers, to obtain the converted Java code file for a language-verification step; and

b) a language-verification step for verifying said converted Java code file for compliance with Java language specifications.

2. (cancelled)

3. The method for language verification of a Java card CAP file according to Claim 1, wherein said mapping substep is performed using a referenced Java export file which is available as a result of creating said Java card CAP file from said original Java code file.

4. The method for language verification of a Java card CAP file according to Claim 1, further comprising:

c) a signature step for creating, after verification of said converted Java code file in said language verification step, a cryptographic signature file for the Java card CAP file.

5. The method for language verification of a Java card CAP file according to Claim 4, further comprising:

d) a loading step for loading the cryptographic signature file to a chipcard together with the Java card CAP file, wherein the cryptographic signature file is attached to the Java card CAP file when loaded in the chipcard.

6. The method for language verification of a Java card CAP file according to Claim 4, wherein the cryptographic signature file is cryptographically verifiable, said method further comprising:

e) an executing step for executing said Java card CAP file upon a positive cryptographic verification.

7-9. (cancelled)

10. A computer-readable medium embodying computer program code, the computer program code comprising computer executable instructions configured for:

converting said Java card CAP file into a corresponding converted Java code file that is semantically identical to said Java card CAP file;

verifying said converted Java code file for compliance with Java language specifications;

converting Java card IDs contained in said Java card CAP file into symbolic names;

converting said Java card CAP file into a standard Java format, to obtain a preconverted file; and

replacing in said preconverted file externally defined names with original names by using a mapping scheme between Java names and tokenized identifiers, to obtain the converted Java code file.

11. (cancelled)

12. A computer-readable medium containing computer program code for a Java card CAP file language verifier for verifying a Java card CAP file that has been derived from an original

Java code file, said Java card CAP file including original Java semantics of said original Java card file, the computer program code comprising instructions for:

a converter for converting said Java card CAP file into a corresponding converted Java code file that is semantically identical to said Java card CAP file, wherein said converter further includes:

a preconverter for converting Java card IDs contained in said Java card CAP file into symbolic names, and for converting said Java card CAP file into a standard Java format, to obtain a preconverted file; and

a mapper for replacing in said preconverted file externally defined names with original names under use of a mapping scheme, to obtain the converted Java code file; and

a language verifier for verifying said converted Java code file upon its compliance with a Java language specification.

13. (cancelled)

14. The computer-readable medium according to Claim 12, wherein the mapper comprises an input for receiving a referenced Java export file created when a referenced Java card CAP file was converted from its corresponding original Java code file.

15. The computer-readable medium of Claim 12, wherein the instructions are further configured for a signature generator for generating a second cryptographic signature file.

16. The computer-readable medium of Claim 15, wherein the instructions are further configured for loading the second cryptographic signature file and the Java card CAP file to a storage device.

17. A computer-readable medium containing computer program code for a reduced file language verifier for verifying a reduced file that has been converted from an original file, the reduced file maintaining original semantics of the original file, the computer program code comprising instructions for:

a converter for converting said reduced file into a corresponding converted file that is semantically identical to said reduced file, wherein said converter further includes:

a preconverter for converting IDs contained in said reduced file into symbolic names and for converting said reduced file into a standard format, to obtain a preconverted file; and

a mapper for replacing in said preconverted file externally defined names with original names under use of a mapping scheme, to obtain the converted file;

means for determining whether said reduced file complies with a predetermined language specification; and

a language verifier for verifying said converted file upon compliance with the predetermined language specification.

18. (cancelled)

19. The computer-readable medium of Claim 17, wherein said mapper comprises an input for a referenced difference file which is available as a result from a conversion in which a referenced reduced file has been converted from its original file.

20. A method comprising:

converting an original file into a reduced file, wherein the original file contains a class description section and an instruction section, and wherein the reduced file contains a code description section that is based on the class description section, and wherein the reduced file contains a code section that is based on the instruction section, wherein the original file contains classes that are capable of being compiled, and wherein the only executable instructions in the reduced file are applets;

converting the reduced file into a converted file, wherein the reduced file and the converted file are semantically identical;

creating a cryptographic signature for the converted file; and

storing the cryptographic signature and the reduced file in a chipcard, wherein the cryptographic signature verifies that the reduced file was converted by a trusted entity.

21. The method of claim 20, wherein the standard code file is a Java™ file, and wherein the CAP file is designed to be used by a Java™ card.

22. The computer-readable medium of claim 10, wherein the Java card CAP file is created from an original file that contains classes that are capable of being compiled, and wherein the only executable instructions in the Java card CAP file are applets.

EVIDENCE APPENDIX

Other than the Office Action(s) and reply(ies) already of record, no additional evidence has been entered by Appellants or the Examiner in the above-identified application which is relevant to this appeal.

RELATED PROCEEDINGS APPENDIX

There are no related proceedings as described by 37 C.F.R. §41.37(c)(1)(x) known to Appellants, Appellants' legal representative, or assignee.